

Quicksilver Scalable Multicast

Krzysztof Ostrowski
Cornell University
krzys@cs.cornell.edu

Ken Birman
Cornell University
ken@cs.cornell.edu

Danny Dolev
Hebrew University
dolev@cs.huji.ac.il

ABSTRACT

Our work is motivated by a platform we’re building to support a new style of distributed programming, in which users drag and drop live components into live documents, often without needing to write new code. The capability requires a multicast layer that scales in dimensions not previously explored. In particular, live documents generate large numbers of multicast groups with irregular overlap. Traditional reliable multicast protocols were conceived for a single group at a time, and multi-group configurations can trigger costly resource contention. Quicksilver Scalable Multicast¹ (QSM) solves these problems using two kinds of mechanisms. First, we introduce several techniques to aggregate traffic when groups overlap. But we also identify a previously unnoticed linkage between memory footprint and CPU consumption, motivating a second class of techniques that minimize memory use and CPU loads. The resulting system is fast, scales well, and is stable under stress. Moreover, our techniques should be applicable in other high-performance distributed systems.

1. INTRODUCTION

Reliable multicast protocols have a long history in distributed computing settings [9,1,8,13,22,23,36], yet are underutilized in modern platforms. The goal of our effort is to overcome obstacles to such uses. One obstacle relates to the way in which multicast platforms are integrated with the programming environment. Reliable multicast has typically been provided either as a library (as in the systems cited above), or as a topic-based publish-subscribe or event notification service [9,25,14]. Vendors perceive both options as prone to misuse, hence in many settings where the technology plays an important role it is hidden from users, for example in IBM’s Websphere platform (business logic state replication [12]) and Microsoft’s Enterprise Cluster Server (fault-tolerant state management [15]).

To make multicast easier to use, we’re exploring a new way of embedding it into distributed settings. The approach is motivated by Web mashup technologies, particularly as supported in .NET, which lets users build applications in a drag and drop fashion, combining pre-made components using Object Linking and Embedding (OLE). QSM extends OLE with a layer that adds support for “live

distributed objects” [27]. Live objects represent distributed functionality such as video streams, replicated files, data structures and services, documents that can be collaboratively edited, chat sessions etc. Users drag live objects onto the desktop or into a “live document”, customize their properties and then share the resulting applications, much in the way one builds a slide set that includes clip-art. When copies of a live document are shared among multiple users, or when users share the same live object in different contexts, a multicast group is formed to propagate updates between its members.

We expect this approach to find broad applicability, for example in financial trading, online gaming, collaboration, and even disaster response. Live objects permit untrained users to quickly create distributed applications, which will share QSM’s robustness – a simple form of scalable reliability today, but stronger guarantees of reliability and security down the road. While building new kinds of live objects isn’t very difficult, even a small “standard” library of customizable objects should suffice for most purposes.

Live objects place new demands on the underlying communication platform: each machine might access many live objects, hence the underlying group communication system needs to scale well even when large numbers of groups overlap. Existing systems scale poorly well in this respect, and this emerges as a second obstacle to broader use of multicast. Moreover, live objects are not the only class of applications that need scalable multicast. Data-centers often replicate multi-component services for load-balancing and fault-tolerance. Each component may need its own replication group, and those groups overlap if components run on the same machines.

To overcome this scalability obstacle:

- QSM leverages IP multicast and only delivers messages to legitimate receivers, yet uses relatively few IP multicast addresses.
- We discovered an important connection between memory footprint and multicast performance in large distributed configurations. QSM introduces an architecture optimized to minimize memory used; it scales smoothly to hundreds of nodes.
- CPU loads also matter. QSM is designed to keep CPU utilization low even at very high data rates.
- We identified causes of throughput oscillations (“multicast storms”). QSM incorporates highly effective preventative mechanisms.

¹ QSM is available at www.cs.cornell.edu/projects/quicksilver/. With the exception of some unpublished technical reports on our download site, this is the first paper to discuss the architecture and performance of QSM.

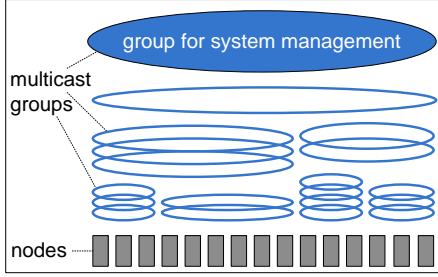


Figure 1. In data centers, where replicated components are shared, multicast groups often overlap to form regular hierarchies.

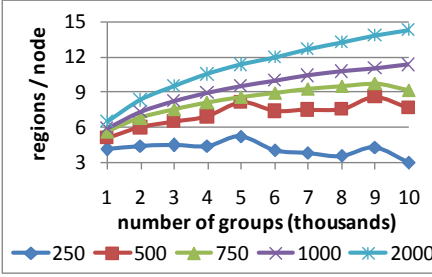


Figure 2. Irregular overlaps decompose into regular “cover sets” constructed of small numbers of regions of overlap.

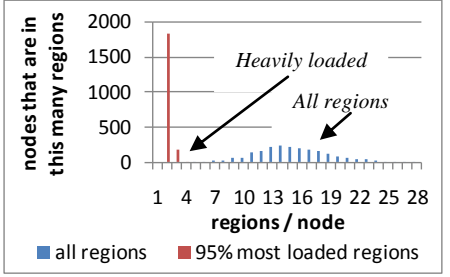


Figure 3. With irregular overlap, most of the traffic seen by a node is concentrated in just 2 of the regions to which it belongs.

- Disruptions happen: nodes join and leave and fail, messages are lost, and non-QSM activities can contend for resources. QSM tolerates disruptions by prioritizing events, using a pull protocol stack architecture, and employing adaptive rate control.

In what follows, we focus on QSM in enterprise computing settings: a LAN or a datacenter. Although we plan to eventually also support WAN and mobile users, brevity prevents us from discussing the associated issues here.

2. PATTERNS OF GROUP OVERLAP

We mentioned two kinds of target settings: datacenters and live objects used on personal workstations. Both result in overlapping communication groups, but the overlap patterns differ: datacenters give rise to *regular* overlap, while live objects will probably yield *irregular* overlap.

We’ll say that multicast groups overlap in a regular way if they can be hierarchically ordered by inclusion on their sets of members, as seen in Figure 1. This type of regularity is common in datacenters, where applications consisting of multiple components are replicated and then deployed within a cluster. If each component needs a group to disseminate updates, the groups overlap because the components are replicated on the same nodes. A hierarchy arises if larger groups are used for control and monitoring purposes, or if larger components are built of smaller ones.

In contrast, widespread use of live objects would probably yield an irregular pattern of group overlap because people’s interests, and hence the documents they access, tend to vary to a large degree. Nevertheless, even in this general case, regularities would sometimes arise. If a live object, such as a video, is shared in different simultaneously active mash-ups, the group corresponding to it could overlap with several other groups, much as in Figure 1.

As will become clear shortly, irregular overlap would pose a problem for QSM (and for many other existing multicast platforms), but there turns out to be a work-around. We start with an observation: even when overlap is highly irregular, it is likely that there will be underlying patterns that can be exploited. For example, many studies have shown that when large numbers of users track large numbers of “information sources”, the popularity of groups will be Zipf-like [14,21,33]. Traffic is also Zipf-like,

although the high traffic groups are not necessarily the most popular ones. Moreover, the Zipf α parameters are large: 2.5 to 3.5 in these studies. Even if one views financial trading and RSS feeds as extreme cases, it seems likely that live documents would exhibit similar behavior (perhaps with slightly smaller α values).

In work reported separately [5], we’ve developed a simple algorithm that decomposes a single irregular pattern of overlapping groups into a collection of “covering sets”. A cover set is just a set of groups that overlap in a regular way, and can be decomposed into regions of overlap within which nodes have identical group membership (with respect to the groups included into the cover set).

Additionally, our algorithm concentrates 95% of the traffic in the system in just a few cover sets. For example, in Figure 2, nodes (250 to 2000) each join 10% of some set of groups (1,000 to 10,000) using a Zipf popularity with $\alpha=1.5$. After running the algorithm, the average node belongs to between 4 and 14 regions (one per cover set). But if we look more closely, only a few of these are heavily loaded (Figure 3; here, 2000 processes each joined 10% of a set of 10,000 groups). A node that was a member of thousands of groups sees almost all of its traffic in just two regions! In scenarios modeling shared live documents, traffic often can be concentrated even further, down to just a single high-activity region.

Of course, different processes will see different “most-loaded” regions, but the implication is that if we have a multicast system that works well for a single regular pattern of overlap, and the network itself isn’t a bottleneck, we can just run the platform multiple times, once for each cover set. A node joins cover sets that include it – those in which it is a member of some region. Resource contention won’t be an issue because there won’t be many instances, and most of them are nearly idle. Moreover, since a typical node finds itself in just one or two *high traffic* regions, we can focus our evaluation and optimization on the behavior of the system in a single heavily loaded but regular group overlap scenario. If it does well in this case, it will also do well in systems with irregularly overlapping groups.

There has been prior work on the handling of irregular overlap, notably in financial trading systems [35]. However,

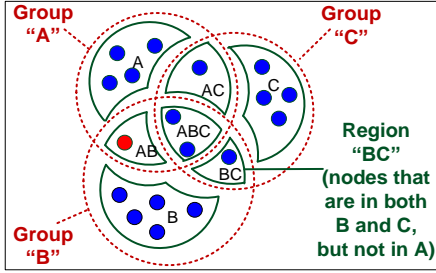


Figure 4. Groups overlap to form regions. Nodes belong to the same region if they are members of the same multicast groups.

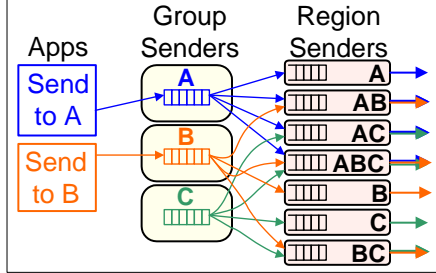


Figure 5. To multicast to a group, QSM sends a copy to each of the regions spanned by the multicast group.

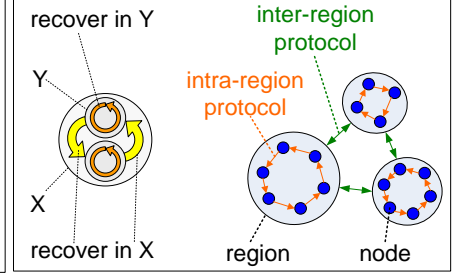


Figure 6. Group span regions, which can be partitioned. A corresponding token ring hierarchy implements loss recovery.

this prior work doesn't concentrate traffic, and it yields "inaccurate" mappings: a node might receive traffic associated with groups to which it doesn't actually belong.

3. PRIOR WORK

The discussion in Section 2 clarifies the challenge: our need is for a reliable multicast system that can sustain high streaming data rates and scales well when potentially large numbers of multicast groups exhibit regular overlap. As a practical matter, since the majority of development today is in "managed" runtime environments such as Java/J2EE and C#/.NET, which standardize memory management, garbage collection and type checking, we also want our system to work well in such settings. As we'll see, the latter requirement has serious implications on the system design.

Reliable multicast is a mature area [17,20,24,28,29], but no existing system would work well under these constraints. First, most multicast systems replicate state within just a single group at a time, for example a single distributed service. Some don't support multiple groups at all, while others run a separate protocol instance per group, and have overheads linear in the number of groups to which a node belongs. Popular toolkits, such as JGroups [2], a core component of the JBoss platform that runs in Java, are designed for, and perform best at, fairly small scales [3] and were not optimized to run at network speeds.

Systems that use IP multicast and run separate protocol instances per group suffer from another problem: with large numbers of IP multicast addresses, the state that needs to be kept by the networking hardware becomes an issue, and we know of datacenters that have abandoned IP multicast based products for this reason. Also, the ability of network adapters at client machines to filter unwanted IP multicast traffic is limited. With hundreds of multicast addresses in use, filtering starts to involve network drivers, which leads to CPU overhead even on machines that haven't subscribed to any of the addresses to which data is being sent.

While systems such as Isis and Spread can support large numbers of "lightweight" groups [1,16,34], the groups seen by applications are an illusion; in actuality, there is just one real group. In Isis, it consists of the union of the members of the lightweight groups. Spread uses a small set

of servers to which client systems connect: each application-level multicast is relayed to one of the servers, multicast in the heavyweight group, filtered at each server depending on whether it has any clients in the lightweight group to which the message was addressed, and unicast by each server to its clients. This approach supports huge numbers of groups with irregular overlap patterns, but at a high cost. In Isis nodes are burdened by undesired traffic in lightweight groups to which they don't belong, while in Spread the servers are a point of contention, and the indirect communication pathway results in high latency.

More recently, application-level multicast systems such as OverCast, NARADA, NICE, or SplitStream [4,10,11,18] have been proposed in place of techniques based on the use of IP multicast. These systems can be remarkably scalable. However, messages follow circuitous routes from source to destination, incurring high latency. In enterprise LANs and datacenters, solutions that do not leverage IP multicast may use networking hardware inefficiently. Moreover, a node can be asked to forward a very high rate of messages that don't interest it, which imposes overheads. These factors are important in our target settings.

4. EXPLOITING REGULAR OVERLAP

Accordingly, we decided to build a new system. Recall from Section 2 that a regular pattern of overlapping groups can be fragmented into one or more *regions* consisting of nodes with exactly the same group membership (Figure 4). A multicast to a single group can now be done by multicasting to each of the regions it spans (Figure 5). In our experiments, no group is ever fragmented into more than 5 regions; the mean was less than 2. Regions generally contained 5 to 10 members and the heavily loaded regions often reflected the intersection of 10 or more groups.

Nodes within a given region receive identical messages, hence they can help each other recover lost messages using a recovery protocol that works for all groups simultaneously and has the overhead independent of the number of groups that overlap on the given region. This offloads work associated with recovery from the sender, avoiding ACK and NAK implosions (the bane of many reliable multicast protocols).

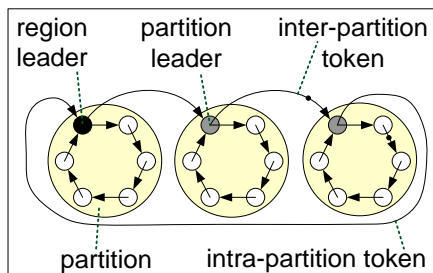


Figure 7. Token rings at higher levels in the hierarchy are run by leader nodes that “represent” entire partitions or regions.

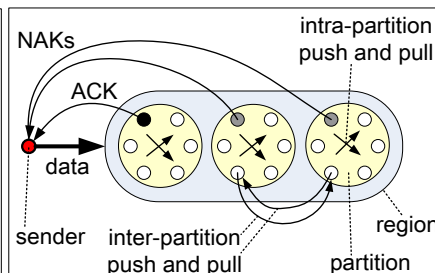


Figure 8. Token rings in partitions enable recovery from nearest neighbors; regional rings enable recovery across partitions.

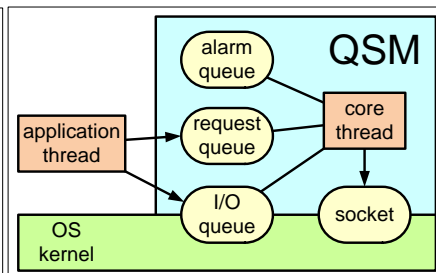


Figure 9. A single QSM thread controls three queues: for I/O events, timer events, and requests from the application threads.

To send a multicast in a region, QSM uses IP multicast; each region is assigned its own multicast address. For any given cover set, a single node will belong to a single region. Thus, as we saw in Figure 3, a single node will only need a few IP multicast addresses even if it belongs to thousands of groups and even if they overlap irregularly. The total number of IP multicast addresses in use is thus small.

To recover from packet loss, multicast groups are subdivided into smaller entities. We have seen how a set of irregularly overlapping groups maps to a small collection of regular cover sets. Within each of those cover sets, a group maps to a set of regions. A large region is further divided into *partitions*. The basic idea will be to perform recovery as locally as possible (Figure 6). QSM does this using a hierarchy of token rings (Figure 7, Figure 8). The lowest level tokens are used within partitions; at this level, each token carries ACK and NAK information about a node. The inter-partition token rings carry information aggregated over entire partitions, and the highest level ring aggregates over regions. (When we tackle huge configurations, it will be necessary to use a deeper hierarchy.)

Each ring triggers some form of “local” recovery. Thus, at the lowest level, a node helps its neighbors recover lost data. If a partition lacks data, its neighbor partitions can help with recovery. Only if no partition has a copy does the sender retransmit the message.

The number and the sizes of the tokens circulating in a region are independent of region size, and of the number of groups mapped to that region, but the size does vary based on the number of distinct senders, and the rate on the volume of traffic in each region. In our experiments, tokens for high-traffic regions are typically 400-800 bytes in size and circulate once a second, even with hundreds of nodes and thousands of groups. The approach keeps QSM control overhead low and almost constant.

To conserve memory, QSM implements *cooperative caching*, similar to a scheme used in Bimodal Multicast [7]. In each region, only one of the partitions retains a copy of any given message, in a round-robin fashion. When a partition lacks a message, a random node within the partition caching that message performs the task. Thus, when bursty loss occurs, retransmissions are usually

performed concurrently. In large scenarios with bursty losses, recovery is remarkably efficient.

The overall system configuration is managed by what we call the Configuration Management Service (CMS), which handles join and leave requests, detects node failures, and uses these to generate a sequence of membership views for each group. The CMS also determines and updates region boundaries, maintains sequences of region views for each region, and tracks the mapping from group views to region views. The CMS is built as a hierarchy of replicated state machines, using a protocol based on that of the Moshe system [19]. Event rates seen by any given level of the hierarchy (joins and failures/departures for which that level is responsible) will be low, hence the CMS shouldn’t limit scalability even in very large deployments.

5. IMPLEMENTATION

QSM is implemented as a .NET component. The entire platform is coded in C# (about 200,000 lines of code), and is accessible from any .NET application. The system is still under development: today it can be used for a single cover set at a time, and the live objects layer is just becoming operational (we’ll release a complete version of QSM by early 2008). The experiments reported here are based on the same version that can be downloaded from our website.

The .NET CLR understands QSM to be the handler for operations on a new kind of event stream. An application can obtain handles to these QSM-managed streams, and can then invoke methods on those handles to send events; incoming messages are delivered through upcalls. QSM is also registered as a “shell extension”, making it possible to access the communication subsystem directly from the Windows GUI. This is the key to one aspect of the live objects concept: a user can store a shortcut to a QSM stream in the file system, and can then point and click on the object or drag it into a live document.

The internal architecture is shown in Figure 9. The system is single-threaded and event-driven. We use a single Windows I/O completion port, henceforth referred to as an “I/O” queue, to collect asynchronous I/O completion events such as notifications of any received messages, completed transmissions, and errors, for all sockets. A “core thread” synchronously polls the I/O queue to retrieve incoming

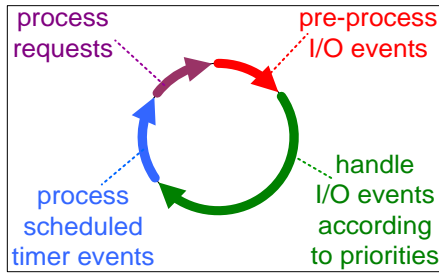


Figure 10. QSM uses time-sharing with a fixed quanta per event type. I/O in QSM is handled in 2 stages, a bit like interrupts.

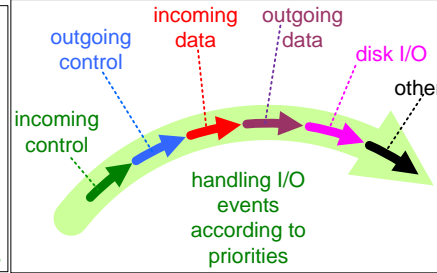


Figure 11. QSM assigns priorities to types of I/O events: control packets or inbound network I/O are handled more urgently.

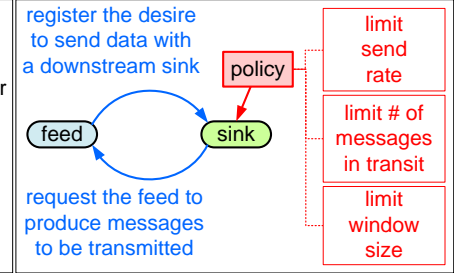


Figure 12. In a pull architecture, data to send is produced in a lazy fashion, when a downstream component is ready to send.

messages. The core thread also maintains an “alarm” queue, implemented as a splay tree, for timer-based events, and a “request” queue, implemented as a lock-free queue with CAS-style operations, for interactions with the application threads. The core thread polls all queues in a round-robin fashion and processes the events sequentially.

Events of the same type are processed in batches, up to the limit determined by a quantum (currently 50ms for I/O, and 5ms for timer events; there is no limit for application requests). When an I/O event representing a received packet is retrieved for a given socket, the socket is drained of any other outstanding I/Os to minimize the probability of loss.

Several aspects of the architecture have significant performance implications. First, QSM is single-threaded and has its own scheduling policy (Figure 10). The pros and cons of using threads in event-oriented systems are hotly debated, but in our case, preemptive multithreading turned out to be not only a major source of overhead due to context switches, but even more importantly, a source of instabilities and oscillatory behaviors, because events were processed in a random order. Eliminating threads let us take control of this order, which greatly improved performance.

QSM prioritizes the incoming I/O over sending-related events to reduce packet loss, and it prioritizes the control packets over data to reduce delays in reacting to packet loss and to reduce the amount of redundant recovery traffic. In Section 5 we’ll see that the latency of control traffic is key to minimizing memory overheads, and as a result, has a significant impact on the overall throughput.

The way QSM prioritizes events mimics interrupt handling. When an I/O event occurs, we retrieve all events from the I/O queue, and distribute them among a number of event queues based on their priorities. Then, we process the events in a priority order (Figure 11).

Second, QSM is designed to avoid buffering messages and to delay creating status messages until the moment they are about to be transmitted. Readers who have implemented multicast protocols will know that most existing systems are push-based: some layer initiates a new message at will, and lower layers buffer that message until it can be sent. This makes sense under the assumption that senders often generate bursts of packets; by buffering them, the

communication subsystem can smooth the traffic flow and keep the network interface busy. One consequence is that messages can linger for a while before they are sent. Not only does this increase memory consumption, but if a message contains “current state” information, that state may be stale by the time it finally is transmitted.

In contrast, QSM implements a “pull” architecture. Our original motivation was to reduce staleness by postponing the creation of the control messages until transmission is about to occur. “Just-in-time” information is more accurate, and this makes QSM more stable. Another benefit is that the “pull” architecture slashes buffering overheads, which, as we shall demonstrate, turns out to have an enormous impact on performance.

In QSM each element of a protocol stack acts as a feed that has data to send, or a sink that can send it (Figure 12), and most play both roles (Figure 13). Rather than creating a message and handing it down to the sink, a feed registers the *intent* to send a message with the sink. The message can be created at this time and buffered in the feed, but the creation may also be postponed until the time when the sink polls the feed for messages to transmit. The sink determines its readiness to send based on a control policy, such as rate, concurrency, windows size limitation, and so forth. When the socket at the root of the tree is ready for transmission, messages will be recursively pulled from the tree of protocol stack components, in a round-robin fashion. Feeds that no longer have data to send are automatically deregistered.

6. EVALUATION

Evaluation of QSM could pursue many directions. Here, we focus on scalability, and particularly on the interactions of the protocol with the runtime environment. By managing these factors, QSM achieves excellent scalability: we can saturate our communication network with relatively modest CPU loads, and see only minor degradation as a function of group size (Figure 14) or the number of groups (Figure 41). Below, we’ll explain the origins of these slowdowns. The current QSM implementation is limited to about 400 nodes/group, and to about 10,000 groups. Our experiments suggest that QSM would sustain its high performance throughout this range.

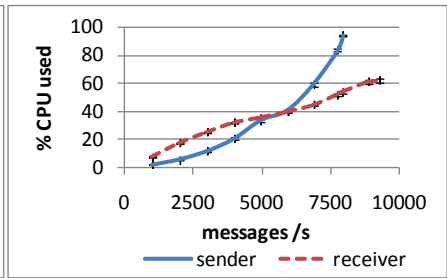
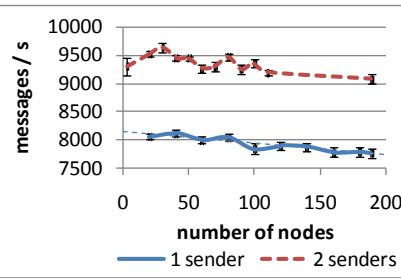
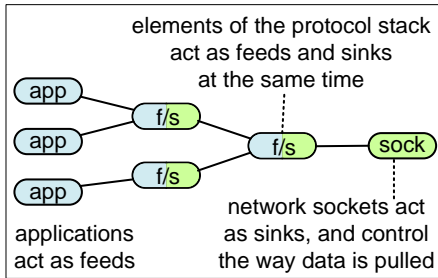


Figure 13. Elements of the protocol stack in QSM form trees rooted at sockets. Each socket “pulls” data from the attached tree. **Figure 14. Max. sustainable throughput in messages/s as a function of the number of receivers with 1 group and 1KB messages.** **Figure 15. CPU utilization as a function of multicast rate (100 receivers).**

Before we get into the details, it may be helpful to summarize our findings. The experiments we report reveal a pattern: in scenario after scenario, the performance of QSM is ultimately limited by overheads associated with memory management in the .NET runtime environment. Basically, the more memory in use, the higher the overheads of the memory management subsystem and the more CPU time it consumes, leaving less time for QSM to run. These aren’t just garbage collection costs: every aspect of memory management gets expensive, and the costs grow linearly in the amount of memory in use. When QSM runs flat-out, CPU cycles are a precious commodity. Thus, in addition to optimizing our code to minimize its direct CPU consumption, minimizing the memory footprint and hence the *indirect* CPU costs were the key to high performance.

These findings aren’t specific to Windows .NET and its CLR. While managed environments such as the .NET CLR do have overheads, we believe the phenomena we’re observing are universal. An application with large amounts of buffered data may incur high context switching and paging delays, and even minor tasks become costly as data structures get large. We will see that memory-related overheads can be amplified in distributed protocols, manifesting as high latency when nodes interact. Since traditional protocol suites buffer messages aggressively, existing multicast systems certainly exhibit such problems, no matter what language they are coded in or what platform hosts them. The mechanisms QSM uses to reduce memory consumption, such as event prioritization, pull protocol stacks, and cooperative caching, should therefore be broadly useful.

The structure of this section is as follows. In 6.1, we show that memory overheads at the sender are linked to protocol latency. In 6.2., we show that similar overheads occur at the receiver, and that latency itself is affected by the overheads it causes. In 6.3 and 6.4, we show that in scenarios with perturbations or when the system is not saturated, the mechanism we identified in 6.1 and 6.2 is still a dominant factor affecting performance. In 6.5 we show that not just the system size, but the number of groups can lead to these sorts of overheads. Finally, in 6.6 we explore behavior if the mechanisms we employed are disabled, or if the system is under stress it cannot handle. The resulting

convoy phenomena and priority inversions destabilize QSM and cause oscillatory behavior.

All results reported here come from experiments on a 200²-node cluster of Pentium III 1.3GHz blades with 512MB memory, connected into a single broadcast domain using a switched 100Mbps network. Nodes run Windows Server 2003 with the .NET Framework, v2.0. Our benchmark is an ordinary application, linked to QSM on the same node. Unless otherwise specified, we send 1000-byte arrays, without preallocating them, at the maximum possible rate, and without batching. Nearly all of the figures include 95% confidence intervals, but these intervals are sometimes so small that they may not always be visible.

6.1 Memory Overheads on the Sender

We begin by showing that memory overhead at the sender is central to throughput. Figure 14 shows throughput in messages/s in two experiments with either 1 or 2 senders multicasting to a varying number of receivers, all of which belong to a single group. With a single sender, no rate limit was used: the sender has more work to do than the receivers and on our clusters, it isn’t fast enough to saturate the network (Figure 15). With two senders, we report the highest combined send rate that the system could sustain without developing backlogs at the senders.

Why does performance decrease with the number of receivers? Let’s focus on a 1-sender scenario. Figure 15 shows that whereas receivers are not CPU-bound, and loss rates in this experiment (not shown here) are very small, the sender is saturated, and hence is the bottleneck. Running this test again in a profiler reveals that the percentage of time spent in QSM code is decreasing, whereas more and more time is spent in *mscorlib.dll*, the CLR (Figure 16). More detailed analysis (Figure 17) makes it clear that the increasing overhead is a consequence of increasingly costly memory allocation (*GCHeap::Alloc*) and garbage collection (*gc_heap_garbage_collect*). The former grows by 10% and the latter by 15%, as compared to 5% decrease of throughput. The bulk of the overhead is the allocation of

² We only have 200 Windows PCs for our experiments today, but Berkeley’s DETER testbed will soon expand, making 500-node experiments possible later this year.

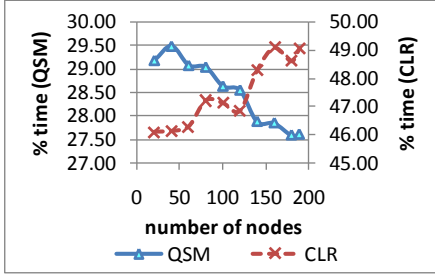


Figure 16. The percentages of the profiler samples taken from QSM and CLR DLLs.

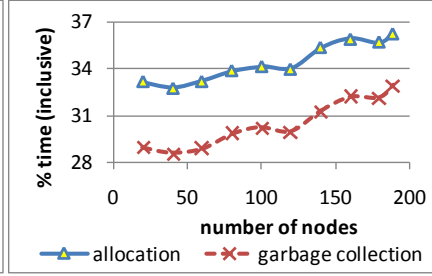


Figure 17. Memory overheads on the sender: allocation and garbage collection.

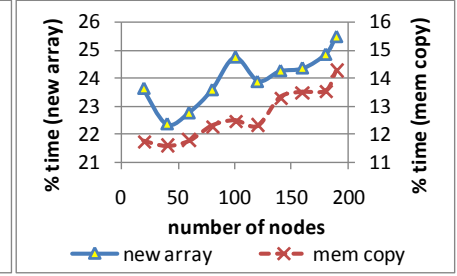


Figure 18. Time spent allocating byte arrays in the application, and copying.

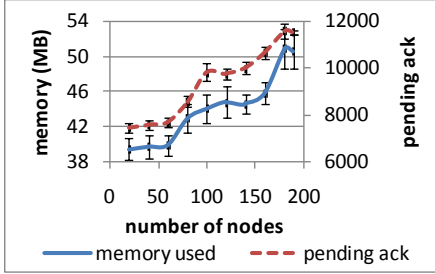


Figure 19. Memory used on sender and the # of multicast requests in progress.

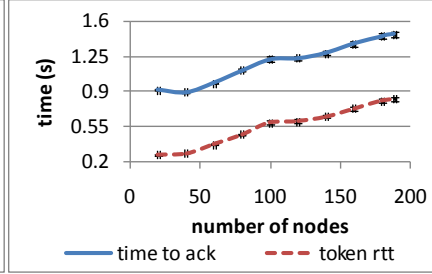


Figure 20. Token roundtrip time and an average time to acknowledge a message.

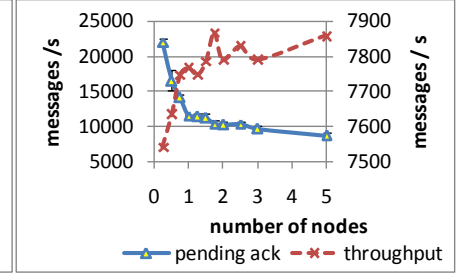


Figure 21. Varying token circulation rate.

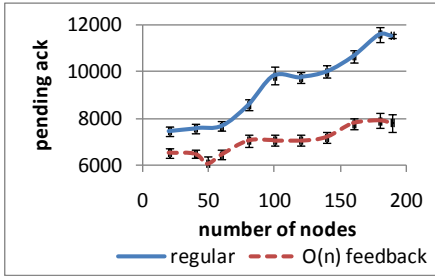


Figure 22. More aggressive cleanup with O(n) feedback in the token and in ACKs.

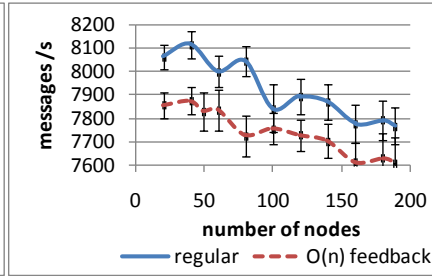


Figure 23. More work with O(n) feedback and lower rate despite saving on memory.

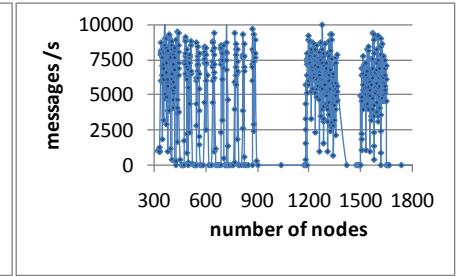


Figure 24. Instability with O(n) feedback.

byte arrays to send in the application (“JIT_NewArr1”, Figure 18). Roughly 12-14% of time is spent exclusively on copying memory internally in the CLR (“memcpy”), even though we used scatter-gather I/O.

The increase in the memory allocation overhead and the activity of the garbage collector are caused by the increasing memory usage. This, in turn, reflects an increase of the average number of multicasts pending ACK (Figure 19). For each, a copy is kept by the sender for possible loss recovery. Notice that memory consumption grows nearly 3 times faster than the number of messages pending ACK. If we freeze the sender node and inspect the contents of the managed heap, we find the number of objects in memory to be more than twice the number of multicasts pending ACK. Although some of these have already been acknowledged, they haven’t yet been garbage collected.

Thus, acknowledgement latency accounts for the decreasing sender performance. Now, let’s shift our focus to the latency: what causes it? The growing amount of unacknowledged data is caused by the increase of the average time to acknowledge a message (Figure 20). This grows because of the increasing time to circulate a token

around the region for purposes of state aggregation (“roundtrip time”). The time to acknowledge is only slightly higher than the expected 0.5s to wait until the next token round, plus the roundtrip time; as we scale up, however, roundtrip time becomes dominant. These experiments show that the performance-limiting factor is the time needed for to aggregate state over regions. Moreover, they shed light on a mechanism that links latency to throughput, via increased memory consumption and the resulting increase in allocation and garbage collection overheads.

A 500ms increase in latency, resulting in just 10MB more memory, inflates overheads by 10-15%, and degrades throughput by 5%. One way to alleviate the problem we’ve identified could be to reduce the latency of state aggregation, by using a deeper hierarchy of rings, letting tokens in each of these rings circulate independently. This would create a more complex structure, but aggregation latency would grow logarithmically rather than linearly. But is reducing state aggregation latency the only option? Of two alternative approaches we evaluated, neither could substitute for lowering the latency of the state aggregation.

Our first approach varies the rate of aggregation by increasing the rate at which tokens are released (Figure 21). This helps only up to a point. Beyond 1.25 tokens/s, more than one aggregation is underway at a time, and successive tokens perform redundant work. Worse, processing all these tokens is CPU-costly. Changing the default 1 token/s to 5 tokens/s decreases the amount of unacknowledged data by 30%, but increases throughput by less than 1%.

Our second approach increased the amount of feedback to the sender. In our base implementation, each aggregate ACK contains a single value `MaxContiguous`, representing the maximum number such that messages with this and all lower numbers are stable in the region. To increase the amount of feedback, we permit ACK to contain up to k numeric ranges, $(a_1, b_1), (a_2, b_2), \dots, (a_k, b_k)$. The system can now clean-up message sequences that have k gaps.

In the experiment shown in Figures 22 and 23, we set the number of ranges proportional to the number of nodes. Unfortunately, while the amount of acknowledged data is reduced by 30%, it still grows, and the overall throughput is actually lower because token processing becomes more costly. Furthermore, the system becomes unstable (notice the large variances in Figure 24), because our flow control scheme, based on limiting the amount of unacknowledged data, breaks down. While the *sender* can now cleanup any portion of the message sequence, *receivers* have to deliver in FIFO order. The amount of data they cache is larger, and this reduces their ability to accept incoming traffic.

6.2 Memory Overheads on the Receiver

The growth in cached data at the receivers repeats the pattern of performance linked to memory. The pattern is similar to what we saw earlier: stress that causes the amount of the buffered data to grow, on any node, is enough to slow everything down.

The reader may doubt that memory overhead on receivers is the real issue, considering that their CPUs are half-idle (Figure 25). Can increasing memory consumption affect a half-idle node? To find out, we performed an experiment with 1 sender multicasting to 192 receivers, in which we vary the number of receivers that cache a copy of each message (“replication factor” in Figure 25). Increasing this value results in a linear increase of memory usage on receivers. If memory overheads were not a significant issue on half-idle CPUs, we would expect performance to remain unchanged. Instead, we see a dramatic, super-linear increase of the token roundtrip time, a slow increase of the number of messages pending ACK on the sender, and a sharp decrease in throughput (Figure 26).

The underlying mechanism is as follows. The increased activity of the garbage collector and allocation overheads slow the system down and processing of the incoming packets and tokens takes more time. Although the effect is not significant when considering a single node in isolation, a token must visit all nodes in a region to aggregate the

recovery state, and delays are cumulative. Normally, QSM is configured so that five nodes in each region cache each packet. If half the nodes in a 192-node region cache each packet, token roundtrip time increases 3-fold. This delays state aggregation, increases pending messages and reduces throughput (Figure 26). As the replication factor increases, the sender’s flow control policy kicks in, and the system goes into a form of the oscillating state we encountered in Figure 24: the amount of memory in use at the sender ceases to be a good predictor of the amount of memory in use at receivers, violating what turns out to be an implicit requirement of the flow-control policy.

6.3 Overheads in a Perturbed System

Another question to ask is whether our results would be different if the system experienced high loss rates or was otherwise perturbed. To find out, we performed two experiments. In the “sleep” scenario, one of the receivers experiences a periodic, programmed perturbation: every 5s, QSM instance on the receiver suspends all activity for 0.5s. This simulates the effect of an OS overloaded by disruptive applications. In the “loss” scenario, every 1s the node drops all incoming packets for 10ms, thus simulating 1% bursty packet loss. In practice, the resulting loss rate is even higher, up to 2-5%, because recovery traffic interferes with regular multicast, causing further losses.

In both scenarios, CPU utilization at the receivers is in the 50-60% range and doesn’t grow with system size, but throughput decreases (Figure 27). In the sleep scenario, the decrease starts at about 80 nodes and proceeds steadily thereafter. It doesn’t appear to be correlated to the amount of loss, which oscillates at the level of 2-3% (Figure 28). In the controlled loss scenario, throughput remains fairly constant, until it falls sharply beyond 160 nodes. Here again, performance does not appear to be directly correlated to the observed packet loss. Finally, throughput is uncorrelated with memory use both on the perturbed receiver (Figure 29) or other receivers (not shown). Indeed, at scales of up to 80 nodes, memory usage actually decreases, a consequence of the cooperative caching policy described in Section 3. The shape of the performance curve does, however, correlate closely with the number of unacknowledged requests (Figure 30).

We conclude that the drop in performance in these scenarios can’t be explained by correlation with CPU activity, memory, or loss rates at the receivers, but that it does appear correlated to slower cleanup and the resulting memory-related overheads at the sender. The effect is much stronger than in the undisturbed experiments; the number of pending messages starts at a higher level, and grows 6-8 times faster. Token roundtrip time increases 2-fold, and if a failure occurs, it requires 2 token rounds before repair occurs, and then another round before cleanup takes place (Figures 31, 32). Combined, these account for the rapid increase in acknowledgement latency.

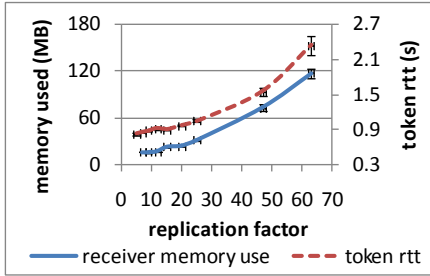


Figure 25. Varying the number of caching replicas per message in a 192-node region.

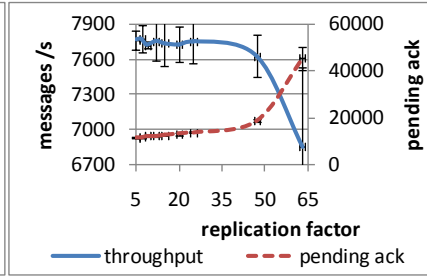


Figure 26. As the # of caching replicas increases, the throughput decreases.

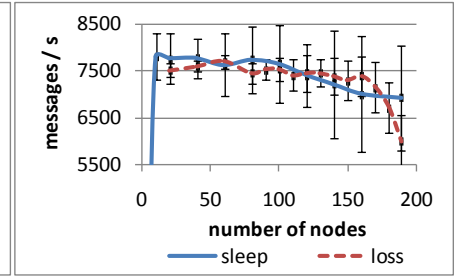


Figure 27. Throughput in the experiments with a perturbed node (1 sender, 1 group).

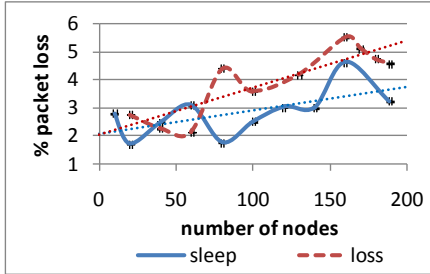


Figure 28. Average packet loss observed at the perturbed node.

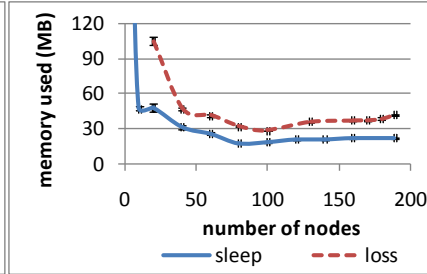


Figure 29. Memory usage at a perturbed node (at unperturbed nodes it is similar).

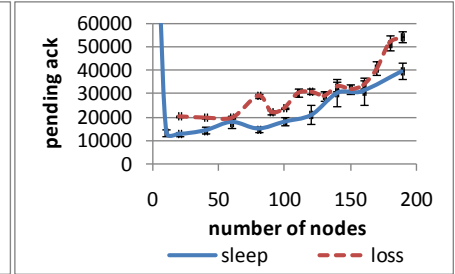


Figure 30. Number of messages awaiting ACK in experiments with perturbances.

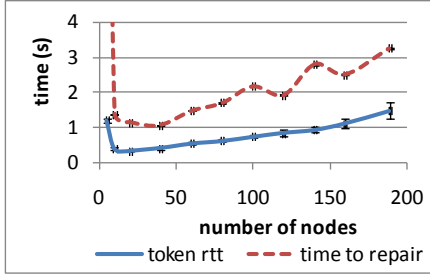


Figure 31. Token roundtrip time and the time to recover in the "sleep" scenario.

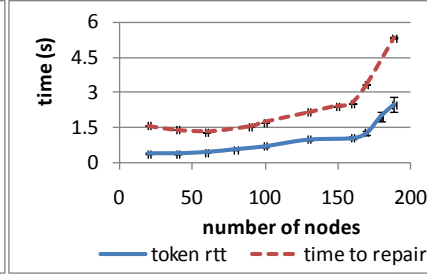


Figure 32. Token roundtrip time and the time to recover in the "loss" scenario.

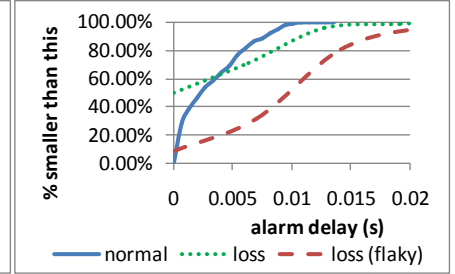


Figure 33. Histogram of maximum alarm delays in 1s intervals, on the receivers.

It's worth noting that the doubled token roundtrip time, as compared to unperturbed experiments, can't be accounted for by the increase in memory overhead or CPU activity on the receivers, as was the case in experiments where we varied the replication factor. The problem can be traced to a priority inversion. Because of repeated losses, the system maintains a high volume of forwarding traffic. The forwarded messages tend to get ahead of the tokens, both on the sending, and on the receiving path. As a result, tokens are processed with higher latency.

Although it would be hard to precisely measure these delays, measuring alarm (timer event) delays sheds light on the magnitude of the problem. Recall that our time-sharing policy assigns quanta to different types of events. High volumes of I/O, such as caused by the increased forwarding traffic, will cause QSM to use a larger fraction of its I/O quantum to process I/O events, with the consequence that timers will fire late. This effect is magnified each time QSM is preempted by other processes on the same node or by the garbage collector; such delays are typically shorter than the I/O quantum, yet longer than the alarm quantum, thus causing the alarm, but not the I/O quanta, to expire.

The maximum alarm firing delays taken from samples in 1s intervals are indeed much larger in the perturbed experiments, both on the sender and on the receiver side (Figures 33 and 34). Large delays are also more frequent (not shown). The maximum delay measured on receivers in the perturbed runs is 130-140ms, as compared in 12-14ms in the unperturbed experiments. On the sender, the value grows from 700ms to 1.3s. In all scenarios, the problem could be alleviated by making our priority scheduling more fine-grained, e.g. varying priorities for control packets, or by assigning priorities to feeds in the sending stack.

6.4 Overheads in a Lightly-Loaded System

So far we've focused on scenarios where the system was heavily loaded, with unbounded multicast rates and occasional perturbations. In each case, we traced degraded performance or scheduling delays to memory-related overheads. But how does the system behave when lightly loaded? Do similar phenomena occur? We'll see that load has a super-linear impact on performance. In a nutshell, the growth in memory consumption causes slowdowns that amplify the increased latencies associated with the growth in traffic.

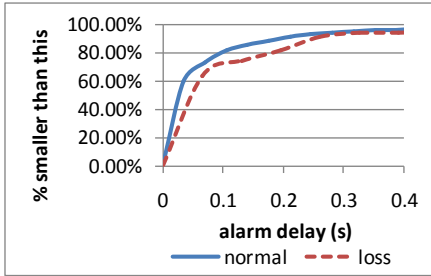


Figure 34. Histogram of maximum alarm delays in 1s intervals, on the sender.

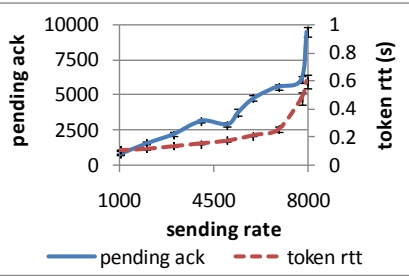


Figure 35. Number of unacknowledged messages and average token roundtrip time on sender and the nearly flat usage on the receiver as a function of the sending rate.

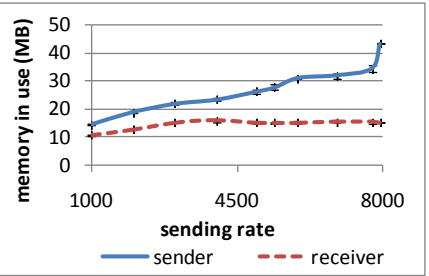


Figure 36. Linearly growing memory use on the sender and the nearly flat usage on the receiver as a function of the sending rate.

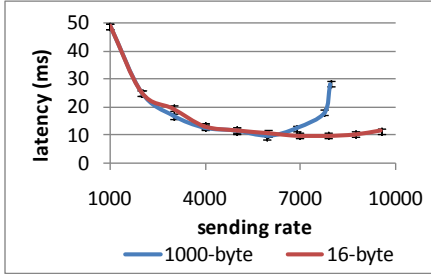


Figure 37. The send-to-receive latency for varying rate, with various message sizes.

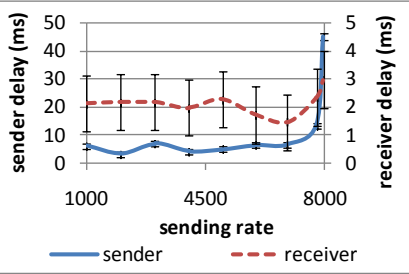


Figure 38. Alarm firing delays on sender and receiver as a function of sending rate.

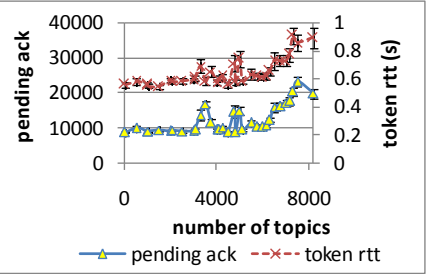


Figure 39. Number of messages pending ACK and token roundtrip time as a function of the number of groups.

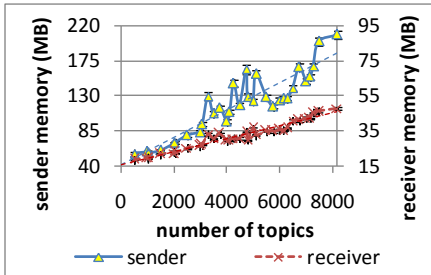


Figure 40. Memory usage grows with the # of groups. Beyond a certain threshold, the system becomes increasingly unstable.

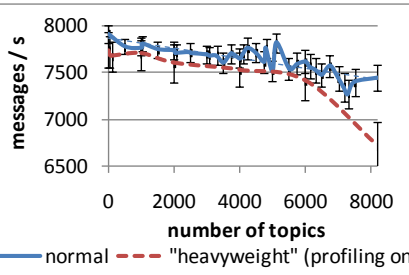


Figure 41. Throughput decreases with the number of groups (1 sender, 110 receivers, all groups have the same subscribers).

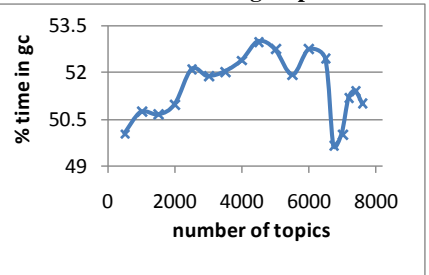


Figure 42. Time spent in the CLR code.

To show this we designed experiments that vary the multicast rate. Figure 15 showed that the load on receivers grows roughly linearly, as expected given the linearly increasing load, negligible loss rates and the nearly flat curve of memory consumption (Figure 36), the latter reflecting our cooperative caching policy. Load on the sender, however, grows super-linearly, because the linear growth of traffic, combined with our fixed rate of state aggregation, increases the amount of unacknowledged data (Figure 35), increasing memory usage. This triggers higher overheads: for example, the time spent in the garbage collector grows from 50% to 60% (not shown here). Combined with a linear growth of CPU usage due to the

increasing volume of traffic, these overheads cause the super-linear growth of CPU overhead shown on Figure 15.

The increasing number of unacknowledged requests and the resulting overheads rise sharply at the highest rates because of the increasing token roundtrip time. The issue here is that the amount of I/O to be processed increases, much as in some of the earlier scenarios. This delays tokens as a function of the growing volume of multicast traffic. We confirm the hypothesis by looking at the end-to-end latency (Figure 37). Generally, we would expect latency to decrease as the sending rate increases because the system operates more smoothly, avoiding context switching overheads and the extra latencies caused by the small amount of buffering in our protocol stack.

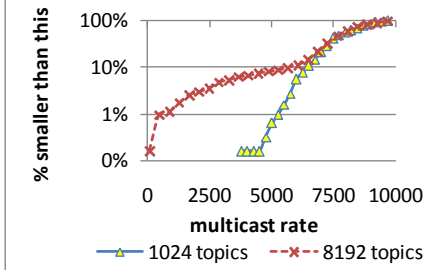


Figure 43. Cumulative distribution of the multicast rates for 1K and 8K groups.

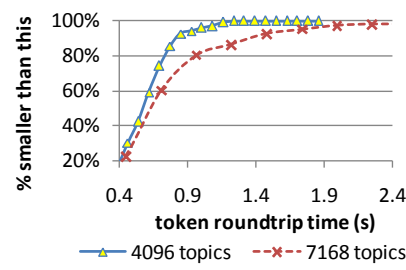


Figure 44. Token roundtrip times for 4K and 7K groups (cumulative distribution).

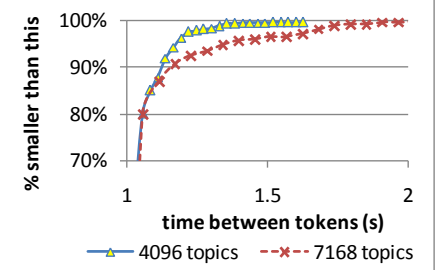


Figure 45. Intervals between subsequent tokens (cumulative distribution).

With larger packets once the rate exceeds 6000 packets/s, the latency starts increasing again, due to the longer pipeline at the receive side and other phenomena just mentioned. This is not the case for small packets (also in Figure 37); here the load on the system is much smaller. Finally, the above observations are consistent with the sharp rise of the average delay for timer events (Figure 38). As the rate changes from 7000 to 8000, timer delays at the receiver increase from 1.5ms to 3ms, and on the sender, from 7ms to 45ms.

6.5 Per-Group Memory Consumption

In our next set of experiments, we explored scalability in the number of groups. A single sender multicasts to a varying number of groups in a round-robin fashion. All receivers join all groups, and since the groups are perfectly overlapped, the system contains a single region. QSM’s regional recovery protocol is oblivious to the groups, hence the receivers behave identically no matter how many groups we use. On the other hand, the sender maintains a number of per-group data structures. This affects the sender’s memory footprint, so we expect the changes to throughput or protocol behavior to be linked to memory usage.

We wouldn’t expect the token roundtrip time or the amount of messages pending acknowledgement to vary with the number of groups, and until about 3500 groups this is the case (Figure 39). However, in this range memory consumption on the sender grows (Figure 40), and so does the time spent in the CLR (Figure 42), hurting throughput (Figure 41). Inspection of the managed heap in a debugger shows that the growth in memory used is caused not by messages, but by the per-group elements of the protocol stack. Each maintains a queue, dictionaries, strings, small structures for profiling etc. With thousands of groups, these add up to tens of megabytes.

We can confirm the hypothesis by turning on additional tracing in the per-group components. This tracing is very lightweight and has no effect on CPU consumption, but it increases the memory footprint by adding additional data structures that are updated once per second, which burdens the GC. As expected, throughput decreases (Figure 41, the “heavyweight” scenario as compared to the “normal” one).

It is worth noting that the memory usage reported here are averages. Throughout the experiment, memory usage

oscillates, and the peak values are typically 50-100% higher. The nodes on our cluster only have 512MB memory, hence a 100MB average (200MB peak) memory footprint is significant. With 8192 groups, peak footprint approaches 360MB, and the system is close to swapping.

Even 3500-4000 groups are enough to trigger signs of instability. Token roundtrip times start to grow, thus delaying message cleanup (Figure 43) and increasing memory overhead (Figure 44). Although the process is fairly unpredictable (we see spikes and anomalies), we can easily recognize a super-linear trend starting at around 6000 groups. At around this point, we also start to see occasional bursts of packet losses (not shown), often roughly correlated across receivers. Such events trigger bursty recovery overloads, exacerbating the problem.

Stepping back, the key insight is that all these effects originate at the sender node, which is more loaded and less responsive. In fact, detailed analysis of the captured network traffic shows that the multicast stream in all cases looks basically identical, and hence we cannot attribute token latency or losses to the increased volume of traffic, throughput spikes or longer bursts of data. With more groups, the sender spends more time transmitting at lower rates, but doesn’t produce any faster data bursts than those we observe with smaller numbers of groups (Figure 43). Receiver performance indicators such as delays in firing timer event or CPU utilization don’t show any noticeable trend. Thus, all roads lead back to the sender, and the main thing affecting the sender is the growing memory footprint.

We have also looked at token round-trip times. The distribution of token roundtrip times for different numbers of groups shows an increase of the token roundtrip time, caused almost entirely by 50% of the tokens that are delayed the most (Figure 44), which points to disruptive events as the culprit, rather than a uniform increase of the token processing overhead. And, not surprisingly, we found that these tokens were delayed mostly on the sender.

With many thousands of groups, the average time to travel by one hop from sender to receiver or receiver to sender can grow to nearly 50-90ms, as compared to an average 2ms per hop from receiver to receiver (not shown). Also, the overloaded sender occasionally releases the tokens with a delay, thus introducing irregularity. For 10%

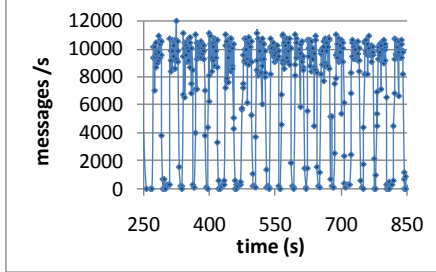


Figure 46. Combined send rate oscillates in 30-sec periods in a 110-node group. The maximum load exceeds receiver capacity.

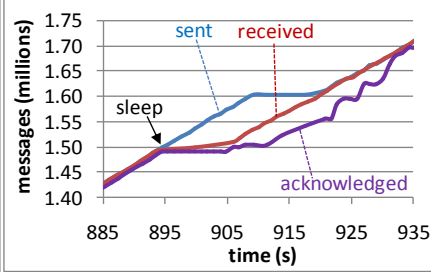


Figure 47. One receiver node “sleeps” for 10s undetected, causing massive recovery. QSM responds by suppressing multicast.

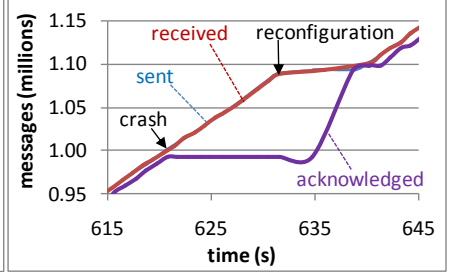


Figure 48. On reconfiguration following a crash, changes can take time to propagate. In this case QSM temporarily slows down.

of the most-delayed tokens, the value of the delay grows with the number of groups (Figure 45). Our old culprit is back: memory-related costs at the sender! To summarize, increasing the number of groups slows the sender, and this cascades to create all sorts of downstream problems that can destabilize the system as a whole

6.6 Oscillatory Behaviors

We like to think of QMS as a crowded highway: the faster it runs, and the shorter the inter-message spacing, the higher the chances of an accident and the more severe the consequences. If a system is too conservative in its handling of loss, failures and the flow control, it fails to achieve the highest speeds. If it is too aggressive, loads can flap from very low to extremely high, causing the kinds of “broadcast storms” that can shut down a datacenter. Our work suggests that oscillating throughput has many causes:

Uncontrolled reaction to failures, for example when a packet is lost by several receivers, stresses the system. The resulting load surge can cause more loss, creating a feedback cycle capable of overwhelming the network (a “broadcast storm”). To avoid such problems QSM does rate-limited recovery triggered (only) by circulating tokens.

Recovery that requires action by a single node, such as a sender, can trigger a kind of convoy in which many nodes must pause until that one node acts – and convoys are contagious because once that node finally acts, other nodes can be overloaded. QSM prevents this via cooperative caching. A burst of losses will often trigger parallel recovery actions by tens of peers.

Jumping the gun by instantly requesting recovery data on the basis of potentially stale state data can trigger redundant work that reinforces the positive feedback loop mentioned earlier. Our “pull” architecture eliminated this issue entirely: we always act upon fresh information.

Priority inversions can leave long lists of messages stacked up waiting for a recovery or control packet. Prioritized event handling is needed to prevent this. Control packets are like emergency vehicles: By letting them move faster than regular traffic, QSM can also heal faster.

Reconfiguration after node joins or failures can destabilize a large system because changes reach different nodes at different times, and structures such as trees or

rings can take seconds to form. QSM suspends multicast and recovery on reconfiguration, and briefly buffers “unexpected” messages, in case a join is underway.

By addressing the problems just mentioned, QSM can stabilize itself in the presence of long bursts of loss or when it experiences artificial “outages” (such as on Figure 27), and tolerates random loss or flakey hardware, responding with reduced throughput. With strong enough perturbation, QSM can still be forced into mild oscillatory behavior. This can be provoked, e.g., by enforcing multicast at a rate exceeding the capacity of the network or of the receivers (Figure 46). Similar behavior is observed with flakey hardware or disruptive applications.

To explore a massive perturbation, we created a “sleep” scenario (recall Figure 27) lasting 10s, causing an 80MB backlog. QSM takes longer time to recover, running recovery at a steady pace, rate controlled, and suppressing multicast until the nodes start to “catch up” (Figure 47). Yet even in such extreme cases QSM can stabilize. Similarly, a reconfiguration following a crash (Figure 48) or join results in slowdown, but the system soon recovers.

7. DISCUSSION

The experiments just reported make it clear that the performance-limiting factor in the QSM system is memory, and that its cost is linked to latency via a positive feedback loop. We believe that our findings are of broad significance. To summarize our design insights:

1. Exploit Structural Regularity. A key enabler to our approach was the recognition that even irregular group overlap can be reduced to a small number of regularly overlapping groups (cover sets), with most of the traffic concentrated in just a few cover sets. This justified a focus on optimizing the regular case and on the performance of QSM in a single, heavily loaded, cover set.

2. Minimize the memory footprint. We expected garbage collection to be costly, but were surprised to realize that when a system has a large memory footprint, the effects are pervasive and subtle. The insight led us to focus on the use of memory throughout our protocols:

(a) **Pull data.** Most multicast systems accept messages whenever the application layer or the protocols produces them. QSM uses an upcall-driven pull architecture. We can delay generating a message until the last minute, and avoid situations in which data piles up in the sender's buffers.

(b) **Limit buffering and caching.** Most existing protocols buffer data at many layers and cache data rather casually for recovery purposes. The overall memory footprint becomes huge. QSM avoids buffering and uses distributed, cooperative caching. This limits message replication and spreads the burden evenly, yet allows parallel recovery.

(c) **Clear messages out of the system quickly.** Data paths should have rapid data movement as a key goal, to limit the amount of time packets spend in the send or receive buffers.

(d) **Message flow isn't the whole story.** Most multicast protocols are optimized for steady low-latency message flow. To minimize memory usage, QSM sometimes accepts increased end-to-end latency for data, so as to allow a faster flow of control traffic, for faster cleanup and loss recovery.

3. Minimize delays. We've already mentioned that the data paths should clear messages quickly, but there are other important forms of delay, too. Most situations in which QSM developed convoy-like behavior or oscillatory throughput can be traced to design decisions that caused scheduling jitter or allowed some form of priority inversion to occur, delaying a crucial message behind a less important one. Implications included the following:

(a) **Event handlers should be short, predictable and terminating.** In building QSM, we struggled to make the behavior of the system as predictable as possible – not a trivial task in configurations where hundreds of processes might be multicasting in thousands of overlapping groups. By keeping event handlers short and eliminating the need for locking or preemption, we obtained a more predictable system and were able to eliminate multithreading, with the associated context switching and locking overheads.

(b) **Drain input queues.** We encountered a tension here: from a memory footprint perspective, one might prefer not to pull in a message until QSM can process it. But in a datacenter or cluster, most message loss occurs in the operating system, not on the network, hence loss rates soar if we leave messages in the system buffers for too long.

(c) **Control the event processing order.** In QSM, this involved single-threading, batched asynchronous I/O, and the imposition of an internal event processing prioritization. Small delays add up in large systems: tight control over event processing largely eliminated convoy effects and oscillatory throughput problems.

(d) **Act on fresh state.** Many inefficiencies can be traced to situations in which one node takes action on the basis of stale state information from some other node, triggering redundant retransmissions or other overheads. The pull architecture has the secondary benefit of letting us delay the

preparation of status packets until they are about to be transmitted, to minimize the risk of such redundant actions.

4. Handle disruptions gracefully. Broadcast storms are triggered when the attempt to recover lost data is itself disruptive, causing convoy effects or triggering bursts of even more packet loss. In addition to the above, QSM employs the following techniques to maintain balance:

(a) **Limit resources used for recovery.** QSM controls recovery traffic rate and delays the creation of recovery packets to prevent them from overwhelming the system.

(b) **Act proactively on reconfiguration.** Reconfiguration takes time. Slowing down and tolerating overheads, buffering packets from “unknown” sources, and delaying recovery to avoid redundant work is a cost worth paying.

(c) **Balance recovery overhead.** In some protocols, bursty loss triggers a form of thrashing. QSM delays recovery until a message is stable on its caching replicas, then coordinates a parallel recovery in which separate point-to-point retransmissions can be sent concurrently by 10s of nodes.

8. FUTURE WORK

Looking to the future, we plan to extend QSM to operate in WAN environments and to support mobile users. Doing so involves non-trivial protocol extensions to tunnel through firewalls and disseminate multicasts. We believe that the system can still scale well in such settings, but much work will need to be done. We're also enhancing QSM's reliability and security properties to include strong semantics such as virtual synchrony or transactional one-copy-serializability and automated data encryption. The reliability layer introduces a novel scripting language (the “Quicksilver Properties Language”) in which these kinds of guarantees can be expressed at a high level [26]. Security will be addressed with per-group keys accessible only to legitimate group members [30,31,32]. Finally, we plan to make the live objects layer extensible. For example, a gossip-based system under development at INRIA/IRISA should eventually also be available in the live objects framework, next to but independent from QSM [6].

9. CONCLUSIONS

The premise of our work is that new options are needed for performing multicast in modern platforms, specifically in support of a new drag-and-drop style of distributed programming inspired by web mash-ups, and for use in enterprise desktop computing environments, or in datacenters where multi-component applications may be heavily replicated. Using multicast in such settings requires a new flavor of scalability - to large numbers of multicast groups - largely ignored in previous work. QSM achieves this by exploiting regularities and commonality of interest.

Our performance evaluations led to a recognition that memory can be surprisingly costly. The techniques QSM

uses to reduce such costs should be useful even in systems that don't run in "managed" environments.

10. ACKNOWLEDGEMENTS

We gratefully acknowledge the comments of our colleagues, including the reviewers of previous drafts of this paper. Ankur Khetrpal, Robbert van Renesse, Einar Vollset and Mahesh Balakrishnan made a number of especially helpful suggestions.

11. REFERENCES

1. Y. Amir, C. Danilov, M. Miskin-Amir, J. Schultz, J. Stanton. The Spread Toolkit: Architecture and Performance. 2004.
2. B. Ban. Design and Implementation of a Reliable Group Communication Toolkit for Java. (1998).
3. B. Ban. Performance tests JGroups 2.5. <http://www.jgroups.org/javagroupsnew/perfnew/Report.html>
4. S. Banerjee, B. Bhattacharjee, and C. Kommareddy. Scalable Application Layer Multicast. ACM SIGCOMM, Aug. 2002.
5. K.P. Birman and K. Ostrowski. Tiling a Distributed System for Efficient Multicast. *Submission to NSDI 2008, technical report available on request.*
6. K. Birman, A.-M. Kermarrec, K. Ostrowski, M. Bertier, D. Dolev, R. Van Renesse. Exploiting Gossip for Self-Management in Scalable Event Notification Systems. ICDCS '07 Workshop on Distributed Event Processing Systems and Architecture (DEPSA). June 2007.
7. K. P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu and Y. Minsky. Bimodal Multicast. ACM Transactions on Computer Systems, Vol. 17, No. 2, pp 41-88, May, 1999.
8. K. P. Birman A Review of Experiences with Reliable Multicast. Software Practice and Experience Vol. 29, No. 9, pp, 741-774, July 1999.
9. K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. Proc 11th ACM SOSP; Austin, Texas, Nov. 1987, 123 – 138.
10. M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron and A. Singh, "SplitStream: High-bandwidth multicast in a cooperative environment", SOSP'03, Lake Bolton, New York, October, 2003.
11. Y. Chu, S. G. Rao, S. Seshan, and H. Zhang. A Case for End System Multicast. IEEE Journal on Selected Areas in Communication (JSAC), Special Issue on Networking Support for Multicast, Vol. 20, No. 8, 2002.
12. E. Decker. http://researchweb.watson.ibm.com/compsci/project_spotlight/distributed/dsc/
13. D Dolev and D Malki. The Transis Approach to High Availability Cluster Communication. CACM 39(4), April 1996, pp 87-92
14. X. Gabaix, P. Gopikrishnan, V. Plerou and H. E. Stanley. A theory of power-law distributions in financial market fluctuations. Nature 423, 267-270 (15 May 2003).
15. R. Gamache. Microsoft Enterprise Cluster Server Architecture. Unpublished presentations, 1999-2001.
16. B. Glade, K. Birman, R. Cooper, and R. van Renesse. Light-Weight Process Groups in the ISIS System. Distributed Systems Engineering, Mar 1994. 1:29-36.
17. M. Handley, S. Floyd, B. Whetten, R. Kermode, L. Vicisano, and M. Luby. The Reliable Multicast Design Space for Bulk Data Transfer, Internet inf. RFC 2887, August 2000.
18. J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, and J. W. O'Toole. Overcast: reliable multicasting with on overlay network. In Proceedings of OSDI'2000.
19. I. Keidar, J. Sussman, K. Marzullo, and D. Dolev. Moshe: A group membership service for WANs. ACM Transactions on Computer Systems, Vol. 20, No. 3, August 2002, p. 191-238.
20. B. N. Levine, and J. J. Garcia-Luna-Aceves. A comparison of reliable multicast protocols. Multimedia Systems 6: 334-348, 1998.
21. H. Liu, V. Ramasubramanian, E.G. Sirer. Characteristics of RSS, A Publish-Subscribe System for Web Micronews. Client and Feed In Proceedings of Internet Measurement Conference (IMC), Berkeley, California, October 2005.
22. X. Liu, C. Kreitz, R. van Renesse, J. Hickey, M. Hayden, K. Birman, and R. Constable. Building Reliable, High-Performance Communication Systems from Components. In Proc. of the 17th ACM SOSP. Dec.1999.
23. S. Maffeis, D. Schmidt. Constructing Reliable Distributed Communication Systems with CORBA. IEEE Communications Magazine feature topic issue on Distributed Object Computing, Vol. 14, No. 2, February 1997.
24. L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, C. A. Lingley-Papadopoulos, T. P. Archambault. The Totem System.. FTCS 25 (1995).
25. B. Oki, M. Pfluegl, A. Siegel, D. Skeen. The Information Bus: An architecture for extensible distributed systems. ACM SOSP, 1993.
26. K. Ostrowski, K. Birman, and D. Dolev. Declarative Reliable Multi-Party Protocols. Cornell University Technical Report, TR2007-2088. March, 2007.
27. K. Ostrowski, K. Birman, and D. Dolev. Live Distributed Objects: Enabling the Active Web. IEEE Internet Computing ICSI-2007-05-0099, Nov/Dec 2007.
28. C. Papadopoulos, and G. Parulkar. Implosion Control For Multipoint Applications. In Proceedings of the 10th Annual IEEE Workshop on Computer Communications, Sept. 1995.
29. S. Pingali, D. Towsley, and J. F. Kurose. A comparison of sender-initiated and receiver-initiated reliable multicast protocols. In SIGMETRICS (1994), pp. 221-230.
30. M. Reiter, K. Birman and R. van Renesse. ACM A Security Architecture for Fault-Tolerant Systems. Transactions on Computer Systems, 12 (4):340-371, Nov 1994.
31. M. Reiter, K. Birman, and L. Gong. Integrating Security in a Group Oriented Distributed System. 1992 IEEE Symposium on Research in Security and Privacy, Oakland, California, May 4-6, 1992, 18-32.
32. M. Reiter and K. Birman. How to Securely Replicate Services. ACM Transactions on Programming Languages and Systems, May 1994, 16(3): 986-1009.
33. B. M. Roehner *Patterns of Speculation: A Study in Observational Econophysics*. Cambridge University Press (ISBN 0521802636). May 2002.
34. L. Rodrigues, K. Guo, P. Verissimo K. Birman. A dynamic light-weight group service. Journal of Parallel and Distributed Computing 60: 12 (Dec 2000), 1449 - 1479
35. Y. Tock, N. Naaman, A. Harpaz, and G. Gershinsky. Hierarchical Clustering of Message Flows in a Multicast Data Dissemination System. PDCS 2005, Phoenix, AZ, USA
36. R. van Renesse, S. Maffeis, and K. Birman. Horus: A Flexible Group Communications System. Communications of the ACM. 39(4):76-83. Apr 1996.